COM1028: Software Engineering

Coursework Assignment

Films4You Analytics System

URN	**6779255**
Username	**si00483**
Date	**29/04/2023**

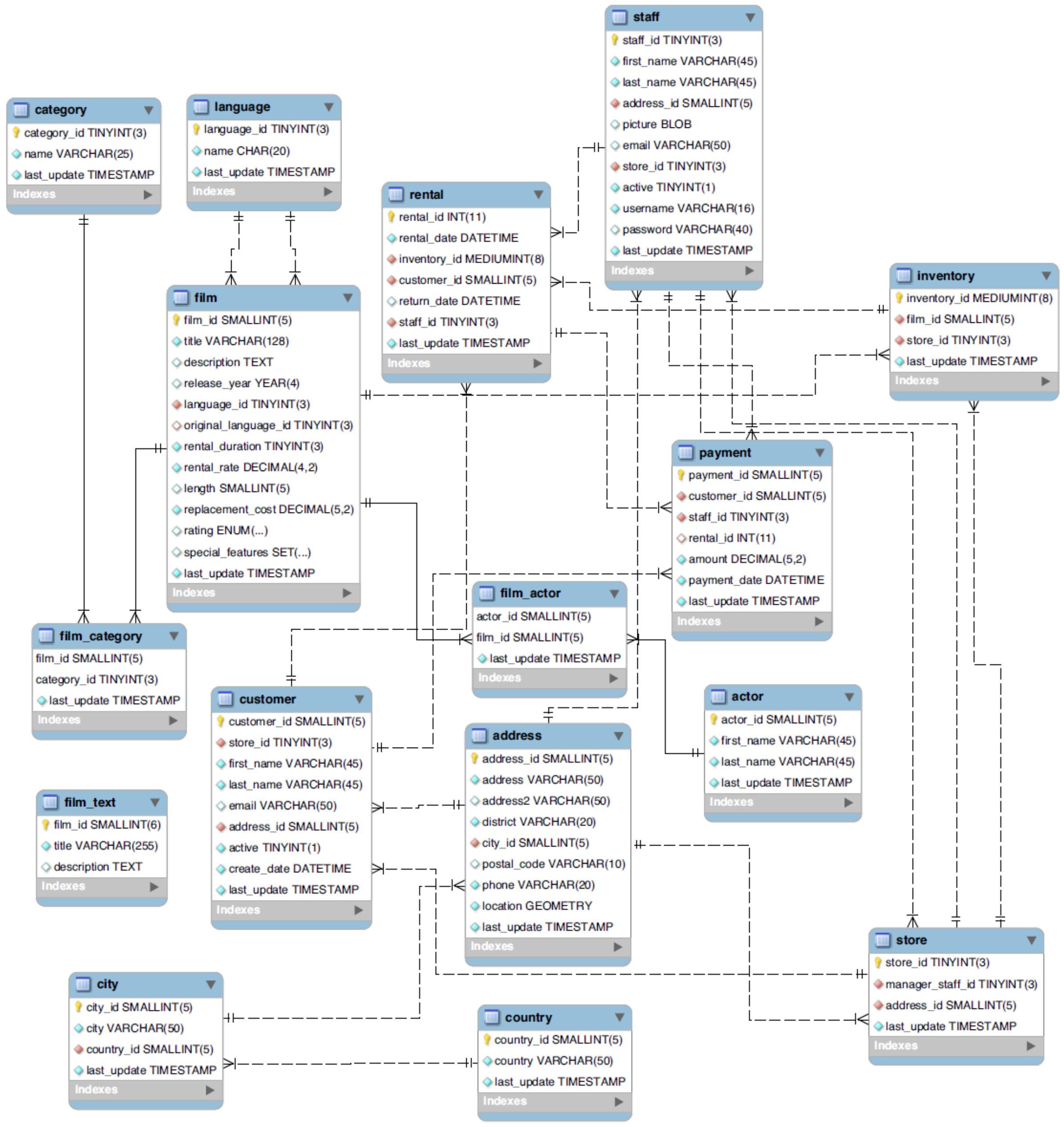
Contents

Introduction

Scenario

"You are working for a client, Films4You, a movie rental company which lends movies to customers. It has a number of staff and physical stores. Films4You wants to build a system that models and interfaces with its existing database. This system should be able to analyse Films4You's database so the company can make informed business decisions to remain competitive. The existing database is used primarily by Films4You's staff, but this could change."

Test Database



Requirement 1 (Q1)

The first requirement is to find the total number of customers using this analytics system.

Proto-personas

Proto-Persona 1

Name: Jon Stephens

Role: Account Executive at Films4You

Age: 32

Background: Jon has a bachelor's degree in business administration and has worked at Films4You for four years. He has experience in sales, customer service, and account management.

Goals:

- Maintain and grow relationships with clients.
- Examine customer data to recognise patterns and preferences.
- Maximise revenue and customer satisfaction.

Challenges:

- Finding the time to analyse customer data thoroughly.
- Understanding customer preferences based on limited data.
- Keeping up with market trends and competition.

Proto-Persona 2

Name: Mike Hillyer

Role: Marketing Manager at Films4You

Age: 38

Background: Mike has a master's in marketing and has worked at Films4You for six years. He oversees the marketing strategy and promotions for the company.

Goals:

- Increase customer acquisition and retention.
- Analyse customer data to create targeted marketing campaigns.
- Optimise marketing budget for maximum ROI.

Challenges:

- Gathering enough data to make informed decisions.
- Effectively targeting different customer segments.
- Staying ahead of competitors in the ever-evolving movie rental industry.

User Stories & Scenario

User Story 1

"As Jon Stephens, the Account Executive, I want to know the total number of customers, as well as the number of active and inactive customers, so that I can evaluate customer growth and identify areas where we can improve our services."

User Story 2

"As Mike Hillyer, the Marketing Manager, I need to find the total number of customers so that I can tailor our marketing campaigns effectively and allocate resources efficiently."

Scenario

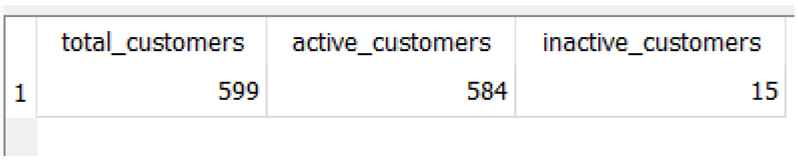
Jon Stephens, an Account Executive at Films4You, uses the new database system to access customer data for his monthly account review. He aims to determine customer growth and pinpoint areas where the company can enhance its services. First, Jon locates the "Customer Overview" section in the system and determines the total number of customers. After analysing the data, Jon notices a decline in customer numbers in the previous quarter. Consequently, he shares this vital information with the marketing team, headed by Mike Hillyer, to work together on a strategy to boost customer acquisition and retention.

Testing

The solution was tested by first using the following SQL statement:

```sql
SELECT
    COUNT(*) AS total_customers,
    COUNT(CASE WHEN active = 1 THEN 1 END) AS active_customers,
    COUNT(CASE WHEN active = 0 THEN 1 END) AS inactive_customers
FROM
    customer;
```

To obtain the customer count, the inactive and active customer counts:



	total_customers	active_customers	inactive_customers
1	599	584	15

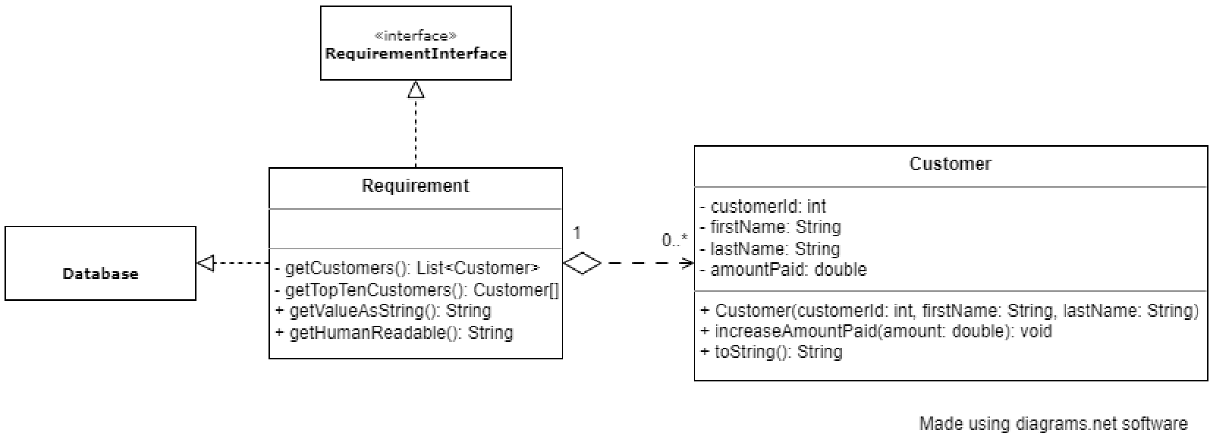
Java unit tests were made using this data for the 'Requirement' class, which implements both the Requirement interface and the test database. These tests check for the correct value as a string and ensure that the human-readable string has the correct format. To test for this, valid and invalid inputs were passed to the class, the accessor outputs were reviewed and expected illegal arguments were made.

(SQL testing was done using 'DB Browser for SQLite')

Requirement 2 (Q2)

The second requirement is to list the top ten customers by revenue.

UML Class Diagram


«interface»
RequirementInterface
Requirement
- getCustomers(): List<Customer>
- getTopTenCustomers(): Customer[]
+ getValueAsString(): String
+ getHumanReadable(): String
Database
1
0..*
Customer
- customerId: int
- firstName: String
- lastName: String
- amountPaid: double
+ Customer(customerId: int, firstName: String, lastName: String)
+ increaseAmountPaid(amount: double): void
+ toString(): String
Made using diagrams.net software

As shown, requirement two has been designed to use two classes: 'Requirement' and 'Customer'. The 'Requirement' class implements the pre-existing interface, 'RequirementInterface', and the database. In addition, it has four methods (two public, two private):

- 'getCustomers()' is a private class that returns the list of customers in the database.
- 'getTopTenCustomers()' is a private class that returns an array of customers, ordered by revenue, where the index 0 stores the customer with the most revenue gained from, using 'getCustomers()'.
- 'getValueAsString()' is a public class that returns a string containing information about the top ten customers, such as their names and IDs, using 'getTopTenCustomers()'.
- 'getHumanReadable()' is a public class that returns the top ten customers as a string in a suitable format so the end user can easily interpret the data using the method 'getValueAsString()'.

The 'Customer' class has four fields (all private) and three methods (all public). The fields are necessary as this class is to model a customer entity from the database. The methods are responsible for constructing the object and handling interactions such as increasing the 'amountPaid' variable (as this will be updated as the database is queried). The 'toString()' method is needed to return all fields of the customer to display its data easily.

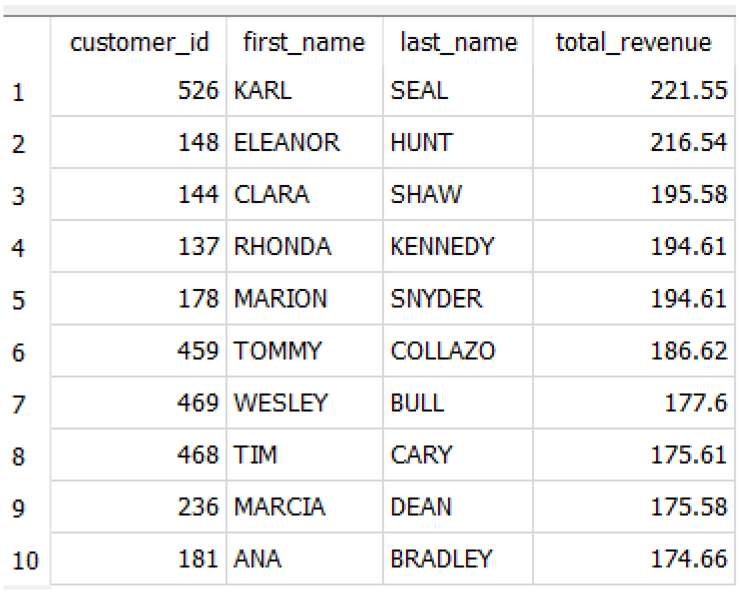
The 'Requirement' class depends on the 'Customer', thus, shares an association. There can be 1 to many 'Customer' instances for one 'Requirement' class but one and only one instance of a 'Requirement' class for one 'Customer' instance. The association is one-way, as only the 'Requirement' class is aware of the existence of the 'Customer' class.

Testing

The solution was tested by first using the following SQL statement:

```sql
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS
total_revenue
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id
ORDER BY total_revenue DESC
LIMIT 10;
```

To obtain the top ten customers by revenue:



	customer_id	first_name	last_name	total_revenue
1	526	KARL	SEAL	221.55
2	148	ELEANOR	HUNT	216.54
3	144	CLARA	SHAW	195.58
4	137	RHONDA	KENNEDY	194.61
5	178	MARION	SNYDER	194.61
6	459	TOMMY	COLLAZO	186.62
7	469	WESLEY	BULL	177.6
8	468	TIM	CARY	175.61
9	236	MARCIA	DEAN	175.58
10	181	ANA	BRADLEY	174.66

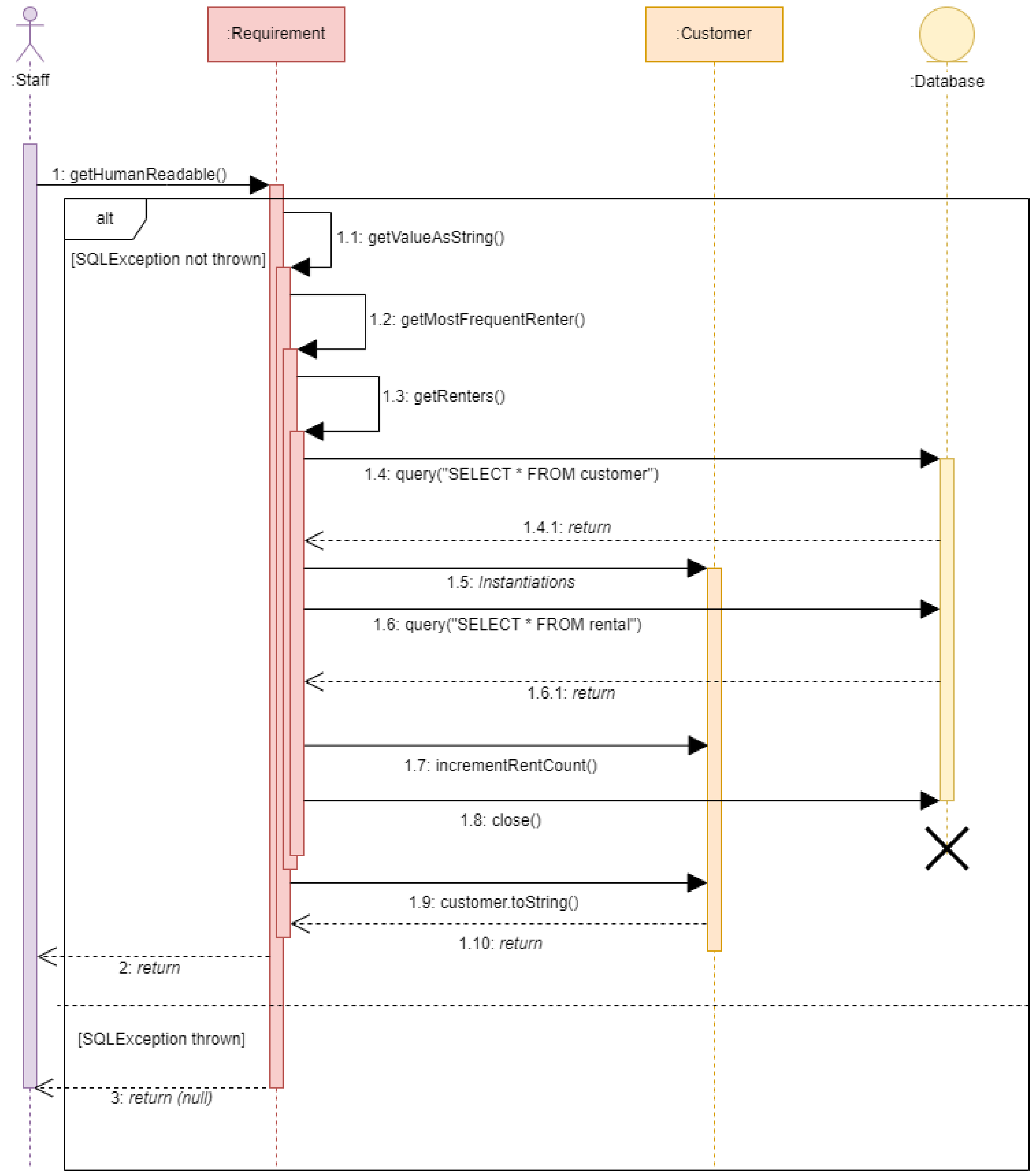
Java unit tests were made using this data for the 'Requirement' class, where these tests check to see if the expected outputs were produced: that the outputs of both 'getValueAsString()' and 'getHumanReadable()' contain the correct data from the test database, and displays them with the expected format. For the 'Customer' class, java unit tests were made to test validation, construction and functionality. To test for this, valid and invalid inputs were passed to the class, the accessor outputs were reviewed and expected illegal arguments were made.

(SQL testing was done using 'DB Browser for SQLite')

Requirement 3 (Q3)

The third requirement is to find the customer who is the most frequent renter.

UML Sequence Diagram



Made using diagrams.net software

As shown in this diagram, as staff members use this method, they interact with multiple classes, methods, and the database:

1. As one of the staff members, the intended user requests to find the customer who is the most frequent renter, causing the 'getHumanReadable()' method to run.

1.1. ‘getHumanReadable()’ calls ‘getValueAsString()’.
1.2. ‘getValueAsString()’ calls ‘getMostFrequentRenter()’.
1.3. ‘getMostFrequentRenter()’ calls ‘getRenters()’.
1.4. 'getRenters()' opens an instance of the database and queries the database with the query "SELECT * FROM customer".
  1.4.1. The database will return all customer entities to 'getRenters()'.
1.5. Using the data from the database, 'getRenters()' creates multiple instances of the 'Customer' class (customers) to model each entity, giving each its name, customer ID and email.
1.6. 'getRenters()' queries the database again with the query "SELECT * FROM rental".
  1.6.1. The database will return all rental entities to 'getRenters()'.
1.7. Using the data from the database, 'getRenters()' updates each customer instance. Hence, each customer instance reflects how often that customer has rented by evaluating customer IDs using the 'incrementRentCount()' method.
1.8. 'getRenters()' closes the database as no more queries are needed and then passes the customers as a list to 'getMostFrequentRenter()', which in turn, seeks through the list to find the most frequent renter and passes the instance to 'getValueAsString()'.
1.9. ‘getValueAsString() will use the customer object’s ‘toString()’.
1.10. The corresponding customer instance will return the stored name, email address and rent count of that customer to 'getValueAsString()', and it will return this information as a string 'getHumanReadable()'.

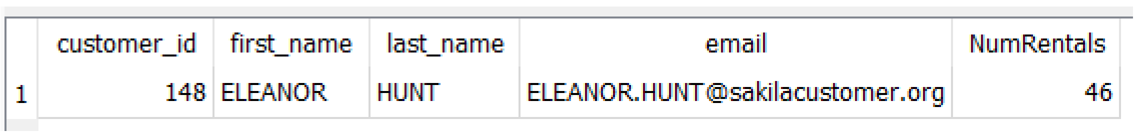
2. 'getHumanReadable()' will then format the string in a suitable format to display this data and return it to the end user.
3. Methods used during this flow can throw an SQL exception, especially when querying the database. If an SQL exception is thrown at any step, the methods will return null to one another until null is received by 'getHumanReadable()', which will return a message to the end user that no results could be found or there was an error.

Testing

The solution was tested by first using the following SQL statement:

```sql
SELECT Customer.customer_id, Customer.first_name, Customer.last_name,
Customer.email, COUNT(*) AS NumRentals
FROM Customer
INNER JOIN Rental ON Customer.customer_id = Rental.customer_id
GROUP BY Customer.first_name, Customer.last_name
ORDER BY NumRentals DESC
LIMIT 1;
```

To obtain the customer with the most rentals:



	customer_id	first_name	last_name	email	NumRentals
1	148	ELEANOR	HUNT	ELEANOR.HUNT@sakilacustomer.org	46

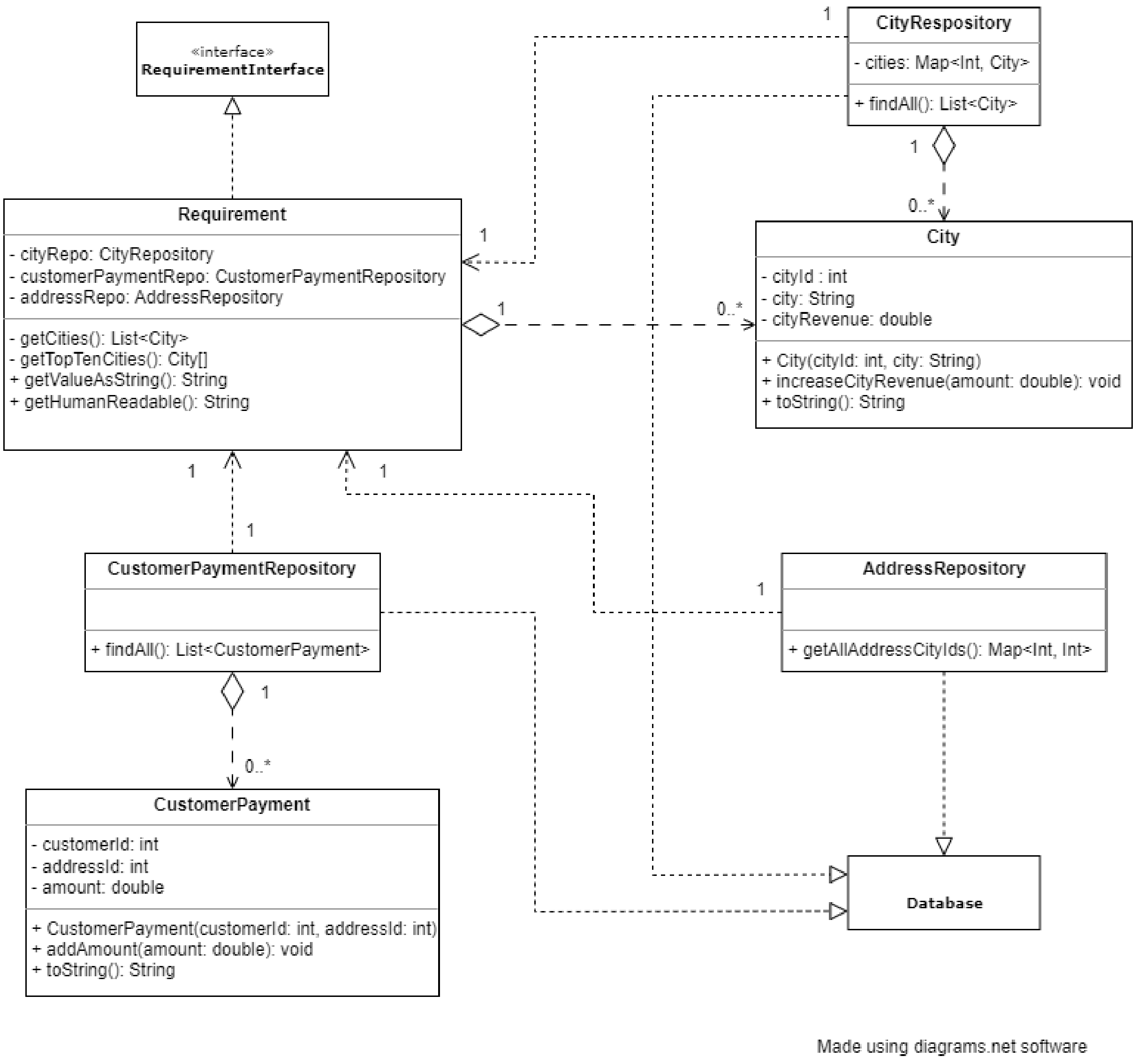
Java unit tests were made using this data for the 'Requirement' class, where these tests check to see if the expected outputs were produced: that the outputs of both 'getValueAsString()' and 'getHumanReadable()' contain the correct data from the test database, and displays them with the expected format. For the 'Customer' class, java unit tests were made to test validation, construction and functionality. To test for this, valid and invalid inputs were passed to the class, accessor outputs were reviewed and expected illegal arguments were made.

(SQL testing was done using 'DB Browser for SQLite')

Requirement 4 (Q4)

This requirement is to list the top ten cities by revenue.

UML Class Diagram


«interface»
RequirementInterface
Requirement
- cityRepo: CityRepository
- customerPaymentRepo: CustomerPaymentRepository
- addressRepo: AddressRepository
- getCities(): List<City>
- getTopTenCities(): City[]
+ getValueAsString(): String
+ getHumanReadable(): String
CityRespository
- cities: Map<Int, City>
+ findAll(): List<City>
City
- cityId : int
- city: String
- cityRevenue: double
+ City(cityId: int, city: String)
+ increaseCityRevenue(amount: double): void
+ toString(): String
CustomerPaymentRepository
+ findAll(): List<CustomerPayment>
AddressRepository
+ getAllAddressCityIds(): Map<Int, Int>
CustomerPayment
- customerId: int
- addressId: int
- amount: double
+ CustomerPayment(customerId: int, addressId: int)
+ addAmount(amount: double): void
+ toString(): String
Database
1
0..*
Made using diagrams.net software

As shown, requirement four has been implemented by using six classes, where the 'City' class is to model city entities in the database, and 'CustomerPayment' class is to model both customer and payment entities, the 'Requirement' class implements the 'RequirementInterface' and 'CityRepository', 'CustomerPaymentRepository' and 'AddressRepository', and all repositories implement the database class which are both pre-existing interfaces and classes, (respectively). Furthermore, requirement four has been modelled in such a fashion to follow the "Repository pattern" (Volkmann, Seppälä and kongleong86, 2015): a design pattern that makes use of repository layers to separate the domain and data mapping layers, which is typically used when accessing databases. Using the repository design pattern, I can avoid duplicate query codes and have distinct classes (repositories) for querying a database, making my solution more maintainable.

Testing

Firstly, the 'CustomerPayment' and 'City' classes were tested by making Java unit tests for validation, construction and functionality. To test for this, valid and invalid inputs were passed to the classes, the outputs from the accessor outputs were reviewed and expected illegal arguments were made.

Secondly, the repositories ('CityRepository', 'AddressRepository' and 'CustomerPaymentRepository') were tested by first querying the databases using SQL statements:

CityRepository SQL Statements

```sql
SELECT COUNT(*) FROM city;

SELECT city FROM city WHERE city_id = 1;

SELECT city FROM city WHERE city_id = 600;
```

AddressRepository SQL Statements

```sql
SELECT COUNT(*) FROM address;
```

CustomerPaymentRepository

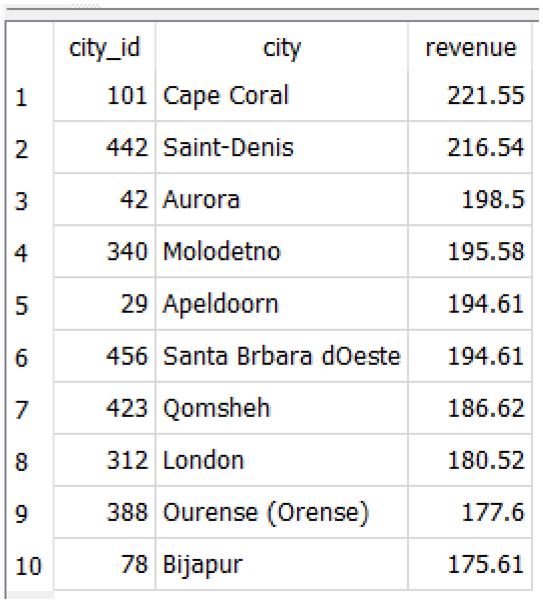
```sql
SELECT SUM(amount) FROM payment WHERE customer_id = 526;

SELECT SUM(amount) FROM payment WHERE customer_id = 181;
```

To obtain the data for testing. As well as the SQL statements, general browsing of the database and previous SQL statements from previous requirements were used to capture this data. After, java unit tests were made to test whether each repository class querying function returned the expected data.

Finally, the 'Requirement class was tested first using the following SQL statement:

```sql
SELECT city_id, city, sum(amount) as revenue
FROM (
SELECT c.city_id, c.city, p.amount, a.city_id AS addr_city_id
FROM city c
JOIN address a ON c.city_id = a.city_id
JOIN customer cust ON cust.address_id = a.address_id
JOIN payment p ON p.customer_id = cust.customer_id
) AS city_revenue
GROUP BY city_id, city
ORDER BY revenue DESC
LIMIT 10;
```

To obtain the top ten cities by revenue:



	city_id	city	revenue
1	101	Cape Coral	221.55
2	442	Saint-Denis	216.54
3	42	Aurora	198.5
4	340	Molodetno	195.58
5	29	Apeldoorn	194.61
6	456	Santa Brbara dOeste	194.61
7	423	Qomsheh	186.62
8	312	London	180.52
9	388	Ourense (Orense)	177.6
10	78	Bijapur	175.61

Java unit tests were made using this data for the 'Requirement' class, where these tests check to see if the expected outputs were produced: that the outputs of both 'getValueAsString()' and 'getHumanReadable()' contain the correct data from the test database, and displays them with the expected format.

(SQL testing was done using 'DB Browser for SQLite')

Critical Analysis and Reflection (Q5)

In this project for an analytics system for 'Films4You', multiple aspects were successful when it came to development, such as:

Code maintainability: Throughout this project, proper steps were taken to ensure the code produced is maintainable such as: adhering to Google's style guide for Java and satisfying the provided check style to produce clean, consistent and easy-to-read code. As well as this, for specific requirements (like requirement four), appropriate design patterns were employed to ensure the code followed industry standard coding conventions and to make the code more maintainable. Also, proper Javadoc commenting was supplied for each class and unit test class, allowing for Javadoc to be generated, furthering better comprehension of the code provided.

Code reliability and testing: By implementing Java unit tests for all provided classes, I ensured the code delivered would be reliable for the live database. The test was made with little-to-no soft-coded elements; therefore, the system will function as intended to meet Films4You's requirements. As well as this, the tests discovered and helped fix issues throughout the development process, making the system even more stable.

System architecture: Using a modular architecture and the skeleton code provided, each requirement belongs to a package, grouping classes into a single API unit on a well-structured foundation. Since the codebase is organised in packages, I worked around the project's complexity effectively, as each package focused on a requirement. Additionally, the packages allowed for code reusability and made it easier to understand the codebase. Also, the skeleton code created a consistent structure, making the code maintainable. This approach to system architecture made integrating requirements for 'Films4You successfully' easier.

However, some things throughout this project could be improved on, such as:

Scalability: Although the codebase was designed using packages which would allow for scalability, requirements such as requirement four use repositories. The repository classes could have used interfaces to make the project more scalable; however, I decided to omit the interfaces as it would be considered a code smell as no new features would be implemented using these repository interfaces. If I had contact with the stakeholders, I could have implemented them to allow for scalability in case the solution needs more features in the future.

Requirement analysis: The project was successful overall since all requirements were implemented; however, the requirements needed to be more extensively defined, leading to assumptions about the definition of aspects such as revenue. In the future, it would be better to obtain more detailed requirements. As well as this, by using proto-personas and having no communication with stakeholders, more assumptions were made for other aspects, such as what information about an entity is displayed and how it is displayed, which may not align with the stakeholders' needs and wants, making my solution less applicable.

Legal and Ethical Compliance: Although industry standards were adhered to, data protection regulations such as GDPR should have been considered in developing the solution. As a result, customers of 'Films4You' may question the solution's reliability and trustworthiness, and the company may face legal action for mishandling data.

To conclude, the Films4You analytics system project succeeded in code maintainability, reliability, testing, and system architecture. However, improvements in scalability, requirement analysis, and legal and ethical considerations are necessary. Addressing these areas will enhance the solution's quality, better meet stakeholder expectations, and result in a more effective analytics system for Films4You.

References

Volkmann, R., Seppälä, I. and kongleong86 (2015). java-design-patterns/repository/. [online] github. Available at: https://github.com/iluwatar/java-design-patterns/tree/master/repository [Accessed 7 May 2023].